BraGuia

March 5, 2024

Diogo Barbosa
PG50326

Tiago Sousa
PG50500

Introdução

Este relatório apresenta uma análise detalhada do projeto BraGuia, uma aplicação desenvolvida visando servir como um guia turístico interativo para a cidade de Braga. O projeto adotou uma abordagem arquitetural baseada no padrão Model-View-ViewModel (MVVM), e implementado com a tecnologia Android Jetpack Compose.

Durante a implementação, foram consideradas diversas soluções de persistência de dados, fluxo de dados unidirecional e integração com serviços externos, como o Google Maps. O relatório discute as funcionalidades implementadas, as limitações encontradas, bem como a gestão de projeto e controlo de versão utilizados ao longo do desenvolvimento.

Detalhes de implementação

Estrutura do projeto

A abordagem arquitetural recomendada usada neste projeto foi a Model-view-viewmodel (MVVM) que separa claramente a lógica de negócio do UI. Na Figura 1 está ilustrada uma representação geral da arquitetura utilizada mais especificamente, tendo em conta que o código foi desenvolvido utilizando o Android Jetpack Compose.

Seguindo a estrutura MVVM, os Composables representam a View, isto é, a secção responsável por apresentar a interface ao utilizador com a informação do ViewModel e enviar para o mesmo os eventos provocados pelas ações do utilizador.

O ViewModel é a fonte de verdade e única fonte de informação do UI. O ViewModel contém um UiState para centralizar a informação pertinente em forma de Flow (semelhante a LiveData), para garantir que sempre que o UiState é atualizado, uma alteração do UI é provocada se necessário.

O Business layer e o Data layer, que inclui Local data (Room Database) e Remote data representam o Model no MVVM. Mais especificamente no nosso projeto a lógica de negócio é feita nos repositórios (Bussiness layer) que obtêm os dados da camada de dados (Data layer), processam-nos e enviam-nos para o ViewModel. A camada de dados neste projeto é composta pelas Data Classes, pelos DAOs, e pela interface que comunica com a *API* (através do Retrofit2). As Data Classes são utilizadas para guardar em memória a informação extraída da base de dados ou da *API* fornecida. Os DAOs são interfaces que declaram as queries feitas à base de dados (geradas automaticamente). Na interface que comunica com a *API* estão declarados os requests HTTP que extraem os dados da *API* fornecida.

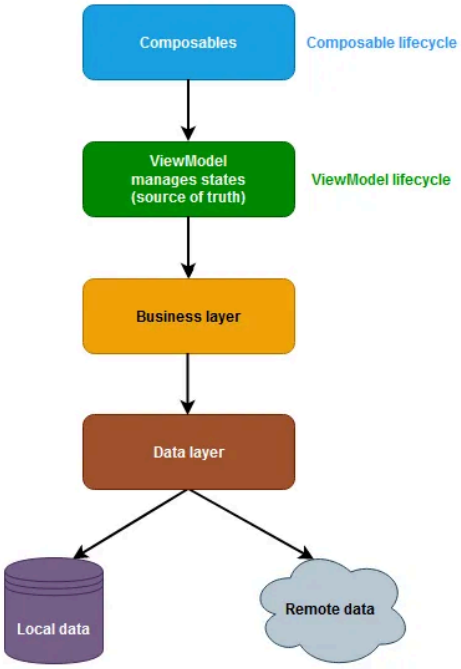

Figura 1: Arquitetura da BraGuia

Soluções de implementação

Persistência de dados

Quanto à persistência de dados na BraGuia foi uma base de dados local *SQLite*. A recolha dos dados é feita primeiramente através da *API* (se possível), que se considera sempre como informação mais atualizada/verdadeira. Estes dados são processados e depois escritos na base de dados, substituindo os antigos, que são considerados desatualizados. A partir desse momento, toda a consulta de informação é feita à base de dados.

A base de dados guarda sempre uma cópia de todos os dados recolhidos da *API*, bem como dados relativos ao utlizador autenticado tal como o histórico e bookmarks. Estes dados relativos ao utilizador permanecem apenas no dispositivo local, podendo ser apagados.

Os dados recolhidos da *API* vêm em formato *JSON* que é mapeado para as nossas Data Classes (através do *GSON*) e segmentado em várias Data Classes para poderem ser introduzidos corretamente numa base de dados relacional. neste caso *SQLite*. Segue-se um exemplo na Figura 2 da classe AppInfo que possui várias relações 1:N, dando origem a várias Data Classes que correspondem a tabelas em *SQLite*:

```
data class AppInfo(

    val appName: String,
    val appDesc: String,
    val socials: List<Social>,
    val contacts: List<Contact>,
    val partners: List<Partner>,
    val landingPageText: String

)
```

Listing 1: AppInfo Data Class

Figura 2: Armazenamento da AppInfo na base de dados

No caso da aplicação não conseguir aceder à *API* por problemas na sua integridade ou problemas de rede, a aplicação ainda é capaz de funcionar. Os dados utilizados serão obtidos exclusivamente da base de dados local, podendo estar desatualizados. No entanto, esta hipótese só é possível se o utilizador já estiver previamente autenticado, visto que não existe forma de autenticar o utilizador sem conexão à internet.

As preferências da aplicação foram implementadas ao nível do dispositivo, ou seja, independentemente do utilizador autenticado, as preferências são as mesmas. Foi usado DataStore para guardar localmente as preferências do dispositivo e aplicá-las ao iniciar a aplicação, como por exemplo o dark/light theme.

Fluxo de dados

O fluxo de dados da BraGuia funciona de forma unidirecional seguindo as recomendações arquiteturais para uma aplicação Android[1]. Os dados fluem da camada de dados para o UI em UiStates através de StateFlows. Um UiState é uma estrutura de dados com a informação da aplicação que é exposta ao UI e é imutável pelo mesmo. O UI apenas faz com que o ViewModel altere este mesmo UiState através de eventos, por exemplo clicar num botão, que depois é refletido no UI.

[1]https://developer.android.com/topic/architecture#unidirectional-data-flow

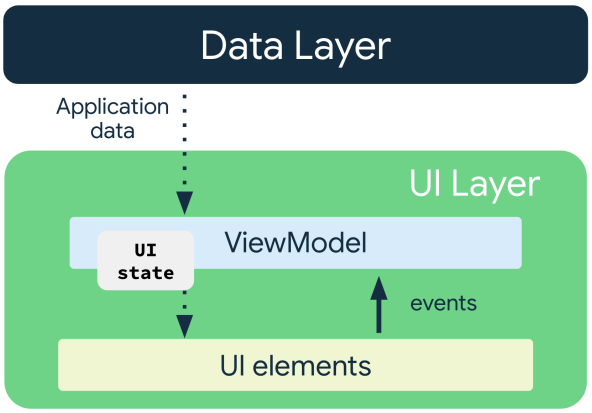

Figura 3: Fluxo de dados da BraGuia

Login/Logout

O login é feito com uma chamada à *API* com o username e password para serem autenticados. Se a resposta for positiva o utilizador entra na aplicação, se não um erro é apresentado. No caso do utilizador já ter sido autenticado previamente e não tiver feito logout, a aplicação verifica localmente se existe algum login feito e garante acesso se assim for o caso, sem necessidade de introduzir novamente credenciais.

O logout é feito eliminando o login guardado localmente e apagando os cookies do utilizador em questão. De seguida o utilizador é levado para a página de login, que requer credenciais para iniciar sessão novamente.

Integração com Google Maps

De modo a mostrar um mapa do roteiro e no ponto de interesse, usamos a biblioteca Maps Compose[2]. Para a navegação do roteiro é usado um Intent com a *API* de navegação do Google Maps[3]

Cookies

Para gerir o uso de cookies usamos a funcionalidade CookieJar do http3, para capturar e injetar nos requests, e CookieManager para guardar e persistir em disco. Exemplo de código Listing 2

Notificações de próximidade

Para notificações de proximidade usamos Geofences, com ajuda do guia oficial do Android[4], onde adicionamos as localizações dos pontos de interesse com um raio de 150 metros. Para podermos notificar o utilizador que uma Geofence foi acionada utilizamos um BroadcastReciever.

Visualização de mídia

De modo a poder visualizar vídeos e clips de áudio na aplicação, usamos a biblioteca Exoplayer[5] e implementamos uma função Composable Media Player. Exemplo de código Listing 3.

[2]https://developers.google.com/maps/documentation/android-sdk/maps-compose
[3]https://developers.google.com/maps/documentation/directions/get-directions#DirectionsRequests
[4]https://developer.android.com/develop/sensors-and-location/location/geofencing
[5]https://developer.android.com/media/media3/exoplayer

Bibliotecas/dependências utilizadas

Retrofit2 e Okhttp3

- implementation("com.google.code.gson:gson:2.8.6")
- implementation("com.squareup.retrofit2:converter-gson:2.9.0")
- implementation("com.squareup.retrofit2:retrofit:2.9.0")
- implementation("com.squareup.okhttp3:logging-interceptor:4.12.0")

Google Maps

- implementation("com.google.maps.android:maps-compose:4.4.0")
- implementation("com.google.android.gms:play-services-maps:18.2.0")
- implementation("com.google.android.gms:play-services-location:17.0.0")
- implementation("com.google.maps.android:maps-ktx:5.0.0")

Datastore

- implementation("androidx.datastore:datastore-preferences:1.0.0")

Exoplayer

- implementation("androidx.media3:media3-exoplayer:1.3.1")
- implementation("androidx.media3:media3-exoplayer-dash:1.3.1")
- implementation("androidx.media3:media3-ui:1.3.1")

Padrões de software utilizados

Singleton: Este padrão é usado quando queremos garantir que uma classe tenha apenas uma instância e forneça um ponto de acesso global a essa instância. Neste projeto este padrão foi utilizado na criação da instância da base de dados.

Factory: O padrão Factory é utilizado quando queremos criar objetos sem especificar a classe exata do objeto que será criado. Em vez disso, usamos uma fábrica que cria o objeto desejado com base num determinado contexto ou condições. Neste projeto este padrão foi utilizado para a criação dos vários ViewModels.

MVVM: Como foi referido anteriormente, usamos o padrão Model View ViewModel.

UDF: Como foi referido anteriormente, quanto ao fluxo de dados, seguimos o *Unidirectional Data Flow*, que dita que numa direção (do Data layer para o UI) flui o UiState e no lado oposto (do UI para o Data Layer) fluem eventos provocam a alteração do mesmo.

Mapa de navegação de GUI

Figura 4: Mapa de navegação

Funcionalidades

Funcionalidades implementadas

- A aplicação deve possuir uma página inicial onde apresenta as principais funcionalidades do guia turístico, descrição, etc.
- A aplicação deve mostrar num ecrã, de forma responsiva, uma lista de roteiros disponíveis;
- A aplicação deve permitir efetuar autenticação;
- A aplicação deve possuir a capacidade de efetuar chamadas para contactos de emergência da aplicação através de um elemento gráfico facilmente acessível na aplicação.
- A aplicação deve assumir que o utilizador tem o Google Maps instalado no seu dispositivo (e notificar o utilizador que este software é necessário);
- A aplicação deve suportar 2 tipos de utilizadores: utilizadores standard e utilizadores premium;
- A aplicação deve guardar (localmente) o histórico de roteiros e pontos de interesse visitados pelo utilizador;
- A navegação proporcionada pelo Google Maps deve poder ser feita de forma visual e com auxílio de voz, de modo a que possa ser utilizada por condutores;
- A aplicação deve ter a capacidade de apresentar e produzir 3 tipos de mídia: voz, imagem e vídeo;
- A aplicação deve possuir uma página de informações acerca do utilizador atualmente autenticado;
- A aplicação deve possuir um menu com definições que o utilizador pode manipular;
- A aplicação deve possuir uma página que mostre toda a informação disponível relativa a um ponto de interesse: localização, galeria, mídia, descrição, propriedades, etc;
- A aplicação deve mostrar, numa única página, informação acerca de um determinado roteiro: galeria de imagens, descrição, mapa do itinerário com pontos de interesse e informações sobre a mídia disponível para os seus pontos;
- A aplicação deve possuir a capacidade de iniciar um roteiro;
- A aplicação deve possuir a capacidade de interromper um roteiro;
- A aplicação deve possuir a capacidade de ligar, desligar e configurar os serviços de localização;

Funcionalidades não implementadas

- A notificação emitida quando o utilizador passa pelo ponto de interesse deve conter um atalho para o ecrã principal do ponto de interesse;
- A aplicação deve possuir a capacidade de descarregar mídia do backend e aloja-la localmente, de modo a poder ser usada em contextos de conectividade reduzida;

Funcionalidades parcialmente implementada

- Para utilizadores premium (e apenas para estes) a aplicação deve possibilitar a capacidade de navegação, de consulta e descarregamento de mídia;
- A aplicação deve possuir a capacidade de emitir uma notificação quando o utilizador passa perto de um ponto de interesse;

Discussão de resultados

Trabalho realizado

Apesar de não termos conseguido implementar todos os requisitos, consideramos que o produto final foi bem concebido, não faltando muito para atingir tudo o que foi pedido. Além disso, o grupo também considera que deveria ter explorado a vertente de testes unitários no código produzido, especialmente tendo em conta a grande complexidade e dimensão do projeto,

Limitações

1. Na implementação do requisito funcional das notificações de proximidade de um ponto de interesse, não conseguimos que o `BroadcastReceiver` reagisse ao evento emitido pela entrada do utilizador na área do ponto de interesse. Isso resultou em não conseguirmos implementar a navegação através da notificação para o ponto de interesse.
2. Na implementação do *logout*, quando executamos a chamada à *API* com o `POST` *logout*, tendo os cookies do utilizador logged in, recebia o código de erro 403 como resposta,e em posteriores chamadas o código inicialmente esperado, 200.
3. Não é possível descarregar mídia.

Funcionalidades extra

Implementamos funcionalidades extras, como os Bookmarks, onde um utilizador pode marcar um roteiro específico, que será então guardado numa lista própria para que o utilizador possa aceder facilmente aos seus roteiros favoritos sempre que desejar. Além disso, introduzimos a opção para o utilizador escolher entre *light* e *dark theme*, permitindo ao utilizador adaptar a interface visual conforme as suas preferências pessoais ou condições de iluminação ambiente.

Gestão de projeto

Gestão e Distribuição de trabalho

- Diogo
  - Base de Dados
  - Model (Data Classes e DAOs)
  - Device Preferences
  - Navegação
- Tiago
  - Integração com Google maps
  - Geofences
  - Media Player
  - Networking
- Ambos
  - UI
  - ViewModel
  - UiState
  - Repositórios

Eventuais metodologias de controlo de versão utilizadas

Para o controlo de versões foi usado o GitHub, permitindo que os membros do grupo trabalhassem em paralelo no código em funcionalidades diferentes, fazendo *merge* e resolvendo eventuais conflitos. Foram utilizadas *git branches* para evitar a introdução de código não funcional na *branch main*.

Conclusão

O projeto BraGuia proporcionou uma valiosa oportunidade de aprendizagem em diversos aspetos do desenvolvimento de aplicações móveis. Ao longo deste processo, enfrentamos desafios que nos permitiram adquirir novos conhecimentos e habilidades, ao mesmo tempo, em que reforçamos conceitos previamente aprendidos.

A adoção da arquitetura MVVM e a utilização do Android Jetpack Compose permitiram uma compreensão mais profunda dos princípios de separação de preocupações e da criação de UIs modernos e reativos. A implementação da persistência de dados local e a integração com serviços externos, como o Google Maps, proporcionaram insights sobre como lidar com dados em diferentes contextos e como utilizar APIs de terceiros de forma eficaz.

Apesar dos obstáculos encontrados, como as dificuldades na integração de notificações de proximidade, esses desafios serviram como oportunidades de aprendizagem, incentivando-nos a procurar soluções criativas e a aprofundar a nossa compreensão dos conceitos envolvidos.

No geral, o projeto BraGuia não só nos permitiu aplicar os conhecimentos teóricos adquiridos em sala de aula, como nos desafiou a expandir as nossas habilidades técnicas, a trabalhar em equipa de forma eficaz e a enfrentar problemas complexos com determinação e criatividade. Estamos confiantes de que as lições aprendidas durante este projeto nos servirão bem no futuro.

Anexos

```
val client = OkHttpClient.Builder()
            .cookieJar(object : CookieJar {
             override fun saveFromResponse(url: HttpUrl, cookies: List<Cookie>) {
                 val cookieManager = CookieManager.getInstance()
                 cookieManager.removeAllCookies(null)
                 for (cookie in cookies) {
                     val cookieString =
                         cookie.name + "=" + cookie.value + "; path=" + cookie.path
                     cookieManager.setCookie(cookie.domain, cookieString)
                 }
                 // saves cookies to persistent storage
                 cookieManager.flush()
             }

             override fun loadForRequest(url: HttpUrl): List<Cookie> {
                 val cookieManager = CookieManager.getInstance()
                 val cookies: MutableList<Cookie> = ArrayList()
                 if (cookieManager.getCookie(url.toString()) != null) {
                     val splitCookies =
                      cookieManager.getCookie(url.toString()).split("[,;]".toRegex())
                             .dropLastWhile { it.isEmpty() }
                             .toTypedArray()
                     for (i in splitCookies.indices) {
                         Cookie.parse(url, splitCookies[i].trim { it <= ' ' })
                             ?.let { cookies.add(it) }
                     }
                 }
                 return cookies
             }
         })
```

Listing 2: Client with cookie managment

```
@Composable
fun MediaPlayer(media: Media) {
    Surface(
        modifier = Modifier
            .fillMaxWidth()
            .height(300.dp), color = Color.Black
    ) {
        val url = media.mediaFile
        val context = LocalContext.current
        val exoPlayer = ExoPlayer.Builder(context).build().apply {
            setMediaItem(MediaItem.fromUri(url))
            repeatMode = Player.REPEAT_MODE_OFF
            playWhenReady = false
            prepare()
        }

        DisposableEffect(
            AndroidView(factory = {
                PlayerView(context).apply {
                    useController = true
                    player = exoPlayer
                }
            })
        ) {
            onDispose { exoPlayer.release() }
        }
    }
}
```

Listing 3: Media Player Implementation